The Intel 64 Architecture

CS520

Dept. of Computer Science
Univ. of New Hampshire

Known generically as x86-64.

Intel 64 is Intel's implementation.

AMD 64 is AMD's implementation.

The design originated with AMD and was later adopted by Intel.

Note: This is not the Intel IA-64 architecture, which is Intel's other 64-bit architecture, for their Itanium processors.

Main Lecture Goal

understand how function calls are supported on the Intel 64.

→ recursion
return address
return values
parameters

necessary in order to understand how to implement garbage collectors, threads, etc.

Intel 64

64-bit addresses

64-bit integer registers

rax, rbx, rcx, rdx, rdi, rsi and r8-r15.

rsp - stack pointer

rbp - frame pointer

rip - instruction pointer (PC)

80-bit floating point registers

internally Intel stores floating-point values in its own non-standard format

values are converted to standard IEEE formats when written to memory

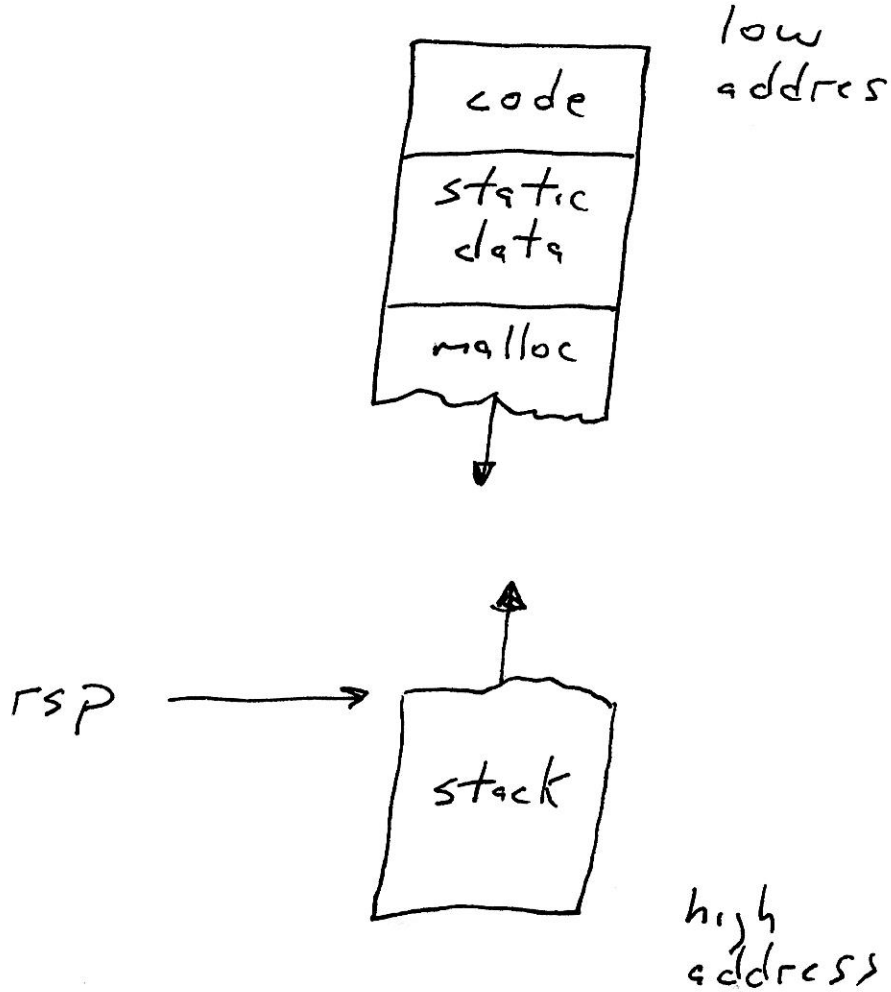
Operand Types

			C types
byte	–	8 bits	b	char
word	–	16 bits	w	short
long	–	32 bits	l	int
quadword	–	64 bits	q	long

stack

grows from high address down to low address

rsp points to the top of stack



Frames

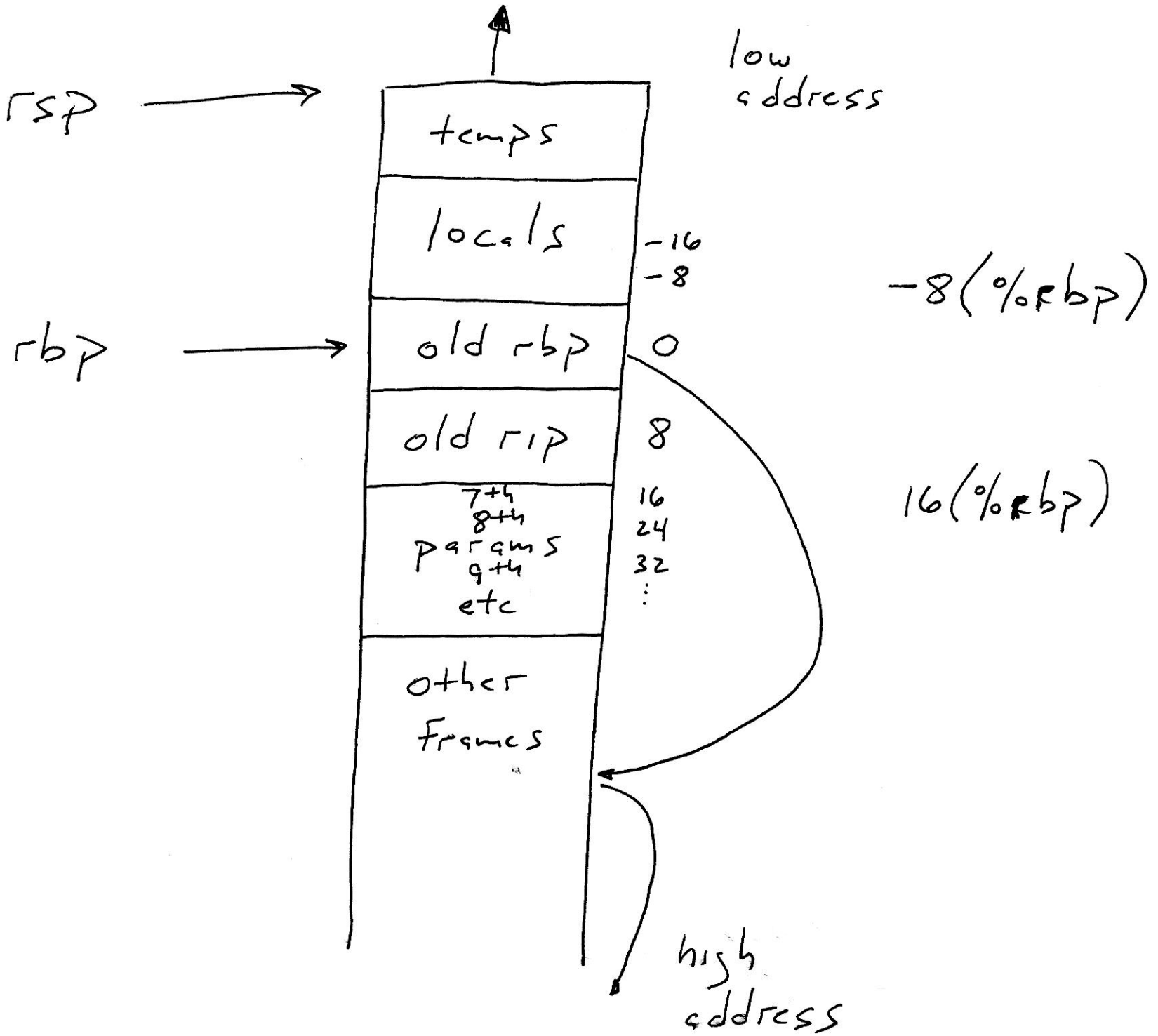
stack contains a series of Frames

one for each function call

each Frame contains:

- return address
- saved registers
- local variables
- parameters
- temporaries

rbp points to the top frame on the stack


low address
rsp
temps
locals
-16
-8
-8(%rbp)
rbp
old rbp
0
old rip
8
16(%rbp)
7th
8th
params
9th
etc
16
24
32
other
Frames
high address

Integer Parameters

1. rdi
2. rsi
3. rdx
4. rcx
5. r8
6. r9
7. ↓ on stack

return value

integer return values are returned in rax

saving/restoring registers

calling function is responsible for saving and restoring rax, rcx, rdx, rdi, rsi, r8-r11 if it needs the old value upon return. — caller saved

→ saved before call & restored after return

called function should save and restore rbx, r12-r15, rsp, rbp if it uses them — callee saved

→ saved upon entry & restored before return

```
#
# x86-64 (Linux) assembler source for computing the factorial function.
#
# The code is based on MIPS code from pages A-26 and A-27 of Patterson &
# Hennessy.
#
# It computes fact(10).
#
    .text
    .align    8
    .globl    main
main:
    pushq     %rbp              # Save old frame pointer
    movq      %rsp,%rbp         # Establish new frame pointer
#
    movq      $10,%rdi          # pass 10 as an argument
    call      fact              # Call factorial function
#
    movq      $.LC0,%rdi        # Pass format string as arg 1
    movq      %rax,%rsi         # Pass return value from fact as arg 2
    call      printf            # Call the printf function
#
    popq      %rbp              # Restore frame pointer
    ret                         # Return to caller

#
    .data
.LC0:
    .string   "The factorial of 10 is %ld\n"
#
#
# The factorial function itself
#
# ie fact(n)
#
    .text
    .align    8
    .globl    fact              # .globl also allows gdb to see label
fact:
    pushq     %rbp              # Save old frame pointer
    movq      %rsp,%rbp         # Establish new frame pointer
    subq      $8,%rsp           # Allocate one local
#
    cmpq      $0,%rdi           # Test n against 0
    jg        .L2               # Branch if n > 0
    movq      $1,%rax           # Return 1
    jmp       .L1               # Jump to code to return
#
.L2:
    movq      %rdi,-8(%rbp)     # Save n into the local
    subq      $1,%rdi           # Compute (n - 1)
    call      fact              # Recursive call
#
    imulq     -8(%rbp),%rax     # Compute n * fact(n - 1)
#
.L1:                            # Result is in %rax
    addq      $8,%rsp           # Deallocate the local
    popq      %rbp              # Restore frame pointer
    ret                         # Return to caller
```

ucs520/public/fact.s

gcc fact.s -o fact

Linux assembler
Intel manuals -
use caution!

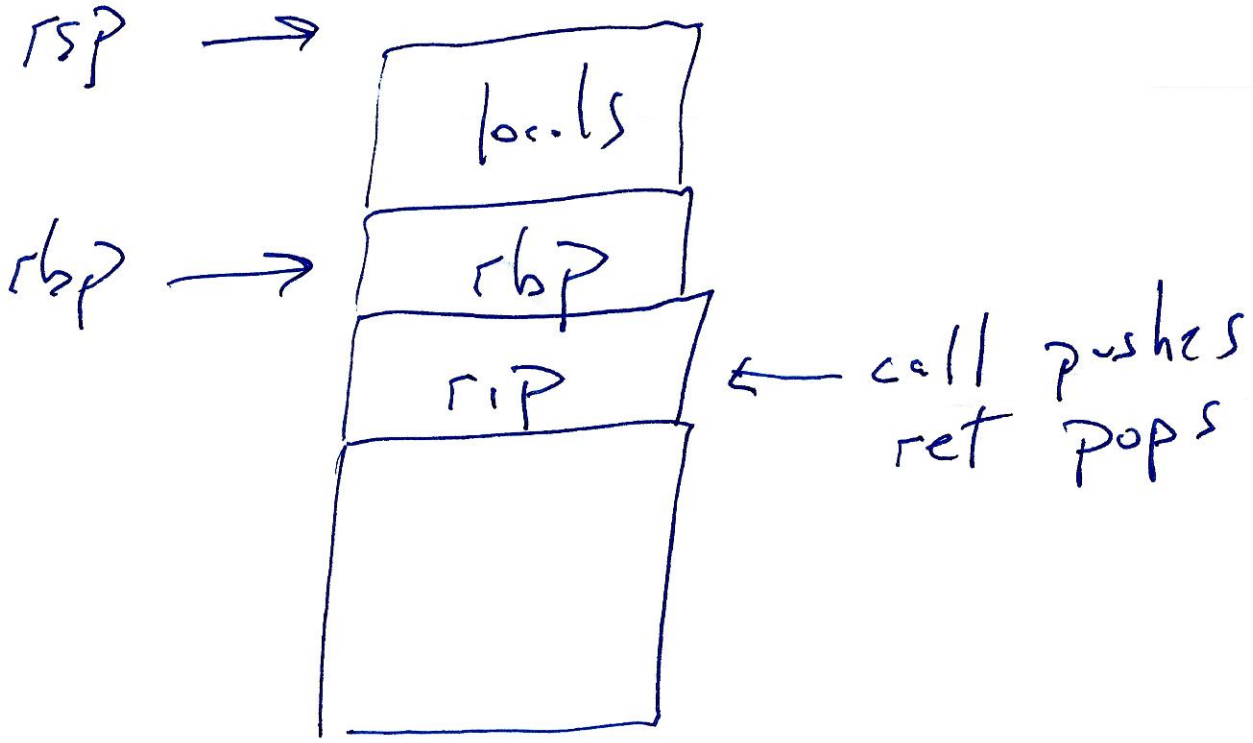
```
#
#
# The factorial function itself
#
# ie fact(n)
#
    .text
    .align   8
    .globl   fact              # .globl also allows gdb to see label
fact:
    pushq    %rbp              # Save old frame pointer
    movq     %rsp,%rbp         # Establish new frame pointer
    subq     $8,%rsp           # Allocate one local
#
    cmpq     $0,%rdi           # Test n against 0
    jg       .L2               # Branch if n > 0
    movq     $1,%rax           # Return 1
    jmp      .L1               # Jump to code to return
#
.L2:
    movq     %rdi,-8(%rbp)     # Save n into the local
    subq     $1,%rdi           # Compute (n - 1)
    call     fact              # Recursive call
#
    imulq    -8(%rbp),%rax     # Compute n * fact(n - 1)
#
.L1:                           # Result is in %rax
    addq     $8,%rsp           # Deallocate the local
    popq     %rbp              # Restore frame pointer
    ret                        # Return to caller
```

text - instructions
data - data

FLAGS register
status register

C
static - hide symbols from other functions
default - visible

asm
.globl - make visible
default - hidden


rsp
rbp
locals
rbp
rip
← call pushes
ret pops

```
#
# x86-64 (Linux) assembler source for computing the factorial function.
#
# The code is based on MIPS code from pages A-26 and A-27 of Patterson &
# Hennessy.
#
# It computes fact(10).
#
    .text
    .align    8
    .globl    main
main:
    pushq     %rbp              # Save old frame pointer
    movq      %rsp,%rbp         # Establish new frame pointer
#
    movq      $10,%rdi          # pass 10 as an argument
    call      fact              # Call factorial function
#
    movq      $.LC0,%rdi        # Pass format string as arg 1
    movq      %rax,%rsi         # Pass return value from fact as arg 2
    call      printf            # Call the printf function
#
    popq      %rbp              # Restore frame pointer
    ret                         # Return to caller
#
    .data
.LC0:
    .string   "The factorial of 10 is %ld\n"
```

printf("The factorial of 10 is %d\n", fact(10));